Boris Wehrle

(E-Mail: boris.wehrle@aitag.com)
ist Consultant bei der AIT AG. Er berät Unternehmen bei der Softwareentwicklung auf Basis von Microsoft Technologien. Seine Schwerpunkte liegen in der Konzeption von verteilten Anwendungen im industriellen Bereich sowie der Unterstützung bei der Umsetzung von Entwicklungsprozessen auf Basis des Visual Studio Team Systems.

CODE-REVIEWS – Jeder kennt's keiner macht's?

Sie sind Softwareentwickler? Sie kennen die Programmierrichtlinien Ihres Unternehmens? Sie halten sich an diese? Auch Ihre Kollegen? Sicher? Architektur- und Programmierrichtlinien unterstützen die Softwareentwicklung im Team und sind eine Basis für die Erstellung von langfristig wartbaren Applikationen. Doch kommen diese auch zur Anwendung? Wird dies regelmäßig überprüft? Verlieren Sie wertvolle Zeit bei Reviews für die Korrektur von immer den gleichen Dingen? Konzentrieren Sie sich bei Code-Reviews auf das Wesentliche. Steigern Sie die Effizienz Ihrer Code-Reviews mit wenigen Schritten.

Die Qualität einer Software kann entsprechend ISO/IEC 9126 anhand einer Reihe von Merkmalen beschrieben werden. Hierzu gehören: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz und Übertragbarkeit. Qualität wird gemessen an den Erwartungen des Kunden. Negative Abweichungen führen zu einer geringeren Nutzerakzeptanz, hohen Aufwänden für Korrekturen und langfristig zu sinkenden Umsätzen. Der Aspekt der Änderbarkeit ist nicht direkt für den Kunden sichtbar. Die Modifizier-, Testbarkeit und Stabilität einer Software bestimmt jedoch entscheidend die Höhe von Entwicklungs- und Testaufwänden.

Da Qualität in eine Software bekanntlich nicht „hineingetestet" werden kann, muss diese geplant, gesichert und fortlaufend kontrolliert werden. Bereits in der Entwurfs- oder Implementierungsphase erkannte Fehler sind deutlich günstiger zu korrigieren, als Fehler, die erst in der Kundeninstallation gefunden werden. Entsprechend sind Investitionen in die Qualitätsprüfung in frühen Projektphasen zumeist effizienter als in späteren. Zusätzlich wirken sich die Investitionen direkt auf die nachfolgenden Phasen aus. Eine Qualitätsinvestition in der Phase Entwurf reduziert den Implementierungsaufwand. Eine Investition in der Implementierungsphase reduziert den Testaufwand.

In den Entwicklungsphasen stehen unterschiedliche Methoden, wie z. B. Prototyping, Unit-Tests und Manuelle Tests, zum Erkennen und Vermeiden von Fehlern zur Verfügung. Die Methoden kommen in der Praxis unterschiedlich häufig zum Einsatz. Eine Entscheidung wird in der Regel auf Basis der entstehenden Kosten (Fehlerverhütungskosten, Prüfkosten) und des zu erwartenden Nutzens (Reduzierung Fehlerkosten) getroffen.

Architektur- und Code-Reviews gehören in der Entwurfs- und Implementierungsphase zu den Erfolg versprechenden Techniken. Softwareartefakte werden entweder gegenseitig oder durch einen erfahrenen Entwickler formell oder informell anhand von unterschiedlichen Kriterien geprüft. Zu diesen gehören unter anderem Verständlichkeit, Funktionalität, Sicherheit und Performance.

Durch die intensive Prüfung kann eine Vielzahl von Problemen frühzeitig erkannt werden. Zusätzlich steigert die intensive Zusammenarbeit der Entwickler die Teamkommunikation. Das fachliche Wissen wird stärker verteilt und somit auch eine Vertretung bei Urlaub oder Krankheit ermöglicht. Gleichzeitig werden unerfahrene Entwickler bzw. neue Teammitglieder geschult bzw. deren Schulungsbedarf erkannt. Die Teamkompetenz steigt. Dennoch kommt die Review-Technik nur selten in der Praxis zur Anwendung. Hierfür gibt es mehrere Gründe:

- Für Reviews werden erfahrene Entwickler benötigt. Diese sind oft bereits durch andere Aufgaben ausgelastet und stehen damit nur eingeschränkt zur Verfügung.
- Werden bei Reviews immer wieder die gleichen Fehler gefunden, sinkt die Motivation der Prüfer weitere Reviews durchzuführen.
- Eine Vielzahl von kleineren Fehlern (z. B. im Bereich Namenskonventionen) verhindert oft das Erkennen von Architektur- und fachlichen Fehlern.
- Der Zeitaufwand und die damit verbundenen Kosten sind hoch.

Dieser Aufwand für Reviews kann durch die Schaffung von entsprechenden Voraussetzungen reduziert werden. Nachfolgend wird zunächst auf Programmierrichtlinien eingegangen. Möglichkeiten einer automatisierten Architekturanalyse werden im Anschluss aufgezeigt.

Programmier- und Architekturrichtlinien

Die Basis für Code-Reviews stellen für das Team verbindliche Richtlinien dar. Diese sollten so knapp wie möglich gehalten werden, um die Akzeptanz bei den Entwicklern

zu erhöhen. Es empfiehlt sich, die Punkte auf die Informationen zu reduzieren, die nicht bereits in Standards (z. B. Microsoft .NET Namenskonventionen) berücksichtigt sind. Abweichungen von Standards sind zu vermeiden. Dies vereinfacht die Zusammenarbeit mit Zulieferern und Freiberuflern. Zwei prägnante Seiten mit Richtlinien, die links und rechts an den Bildschirm geheftet werden können, sind hier deutlich wertvoller als ein umfassendes Dokument, welches im Schrank oder auf der Festplatte verstaubt.

Validierung von Programmierrichtlinien

Die schriftliche Fixierung von Richtlinien kann durch die Verwendung von Tools, die diese automatisch während der Entwicklung prüfen, d. h. einem automatischen Review unterziehen, weiter reduziert werden. Entwickler sind damit nicht gezwungen, Richtlinien auswendig zu lernen, sondern sie werden direkt auf Verstöße und Verletzungen hingewiesen und lernen damit automatisch hinzu.

Statische Code-Analyse

Ein mögliches Werkzeug, um diese Automatisierung zu erreichen, ist das *Visual Studio Team Systems*. Bestandteil der *Visual Studio Team Edition for Software Developers* ist eine Statische Code-Analyse. Der zur Verfügung stehende Regelsatz umfasst mehrere hundert Regeln, unter anderem in den Bereichen Namens-Konventionen, Performance, Sicherheit und Design, und deckt damit auch bereits wichtige Architekturaspekte ab.

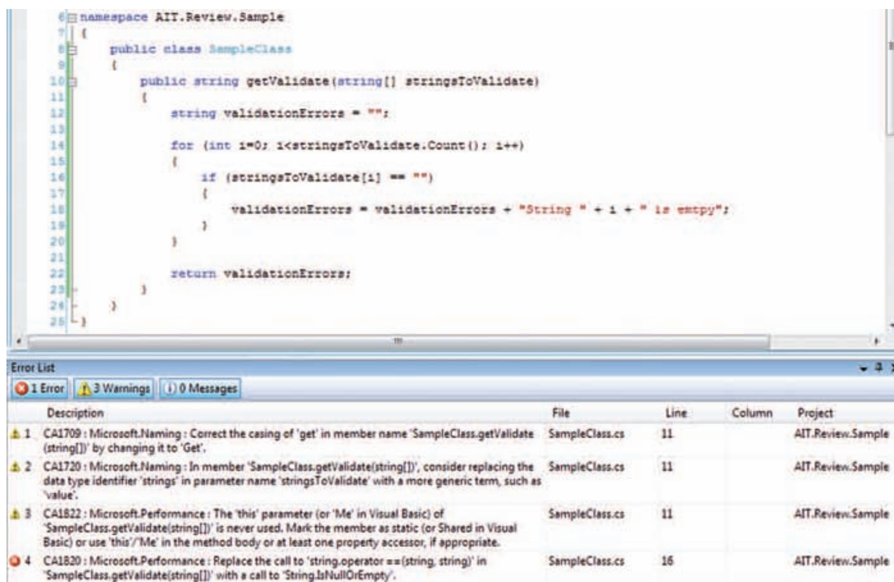

Abb. 1: Verstöße gegen die definierten Regeln werden im Rahmen des Kompiliervorgangs ausgegeben. Über das Kontextmenü wird eine Hilfestellung angeboten.

Die Analyse erfolgt im Rahmen des Kompiliervorgangs. Das Beispiel in Abbildung 1 zeigt, wie Fehler und Warnungen, welche durch Regelprüfung entstehen, dem Entwickler präsentiert werden. Das Beispiel zeigt typische Implementierungen, die zwar syntaktisch richtig sind, sich aber unter .NET eleganter lösen lassen. Insbesondere Umsteiger von VB6 oder C++, die mit den Namenskonventionen und Sprachfeatures noch nicht vertraut sind, profitieren von der Analyse.

StyleCop

Ein weiteres von Microsoft angebotenes freies Add-In ist *StyleCop* [StyMic]. Während die statische Code-Analyse, die formale syntaktische Richtigkeit der Applikation prüft, geht *StyleCop* einen Schritt weiter. Analysiert werden unter anderem Konventionen bei der Klammersetzung, Schreibweisen und die Anzahl der Leerzeilen zwischen Code-Abschnitten. Der Quellcode wird hierdurch weiter vereinheitlicht. Dies vereinfacht erheblich, sich in „fremden" Code einzulesen, um diesen nachzuvollziehen. Insbesondere die Klammersetzung wird in einem Team, das sich aus Entwicklern mit unterschiedlichen Sprachkenntnissen (Java, C++) zusammensetzt fest definiert. Ein Rückfall in alte

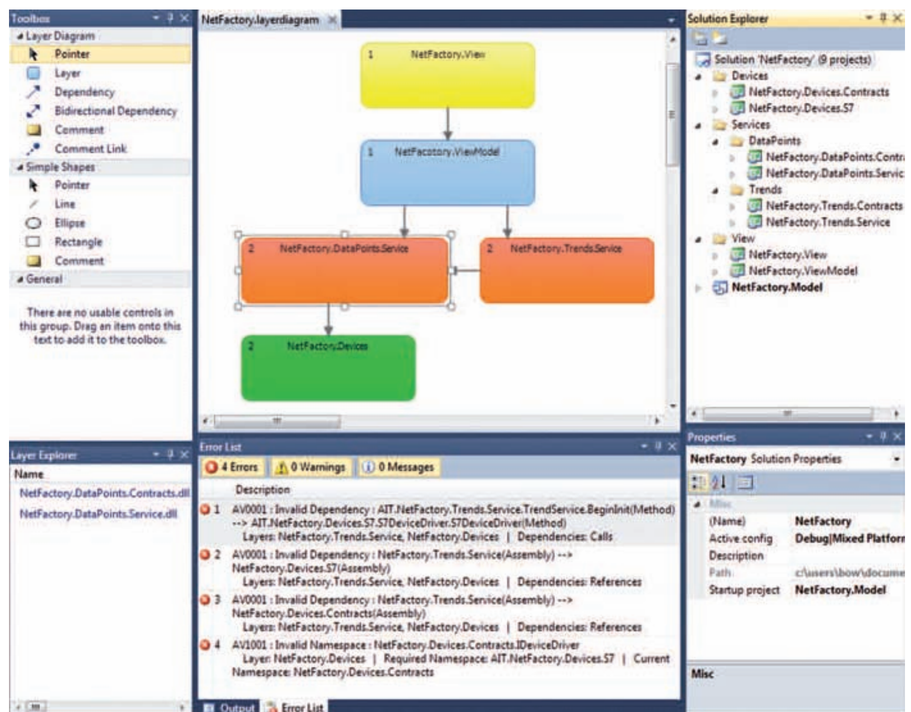

Abb. 2: Modellierung von service- und schichtenorientierten Architekturen.

```
[assembly: AssemblyTitle("Versioning")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyInformation(AssemblyType.Implementation,
    SoftwareLayer.Logic, "Versioning")]
```

Abb. 3: Metadaten für Komponentenarchitekturen.

Gewohnheiten wird wirkungsvoll verhindert. Zusätzlich werden einige architekturrelevante Aspekte, wie z. B. die Verwendung von statischen Methoden, geprüft. Ergebnisse der Analyse werden analog zur Statischen Code-Analyse im Rahmen des Kompiliervorgangs ausgegeben.

Validierung von Architekturrichtlinien

In service- und schichtenorientierten Architekturen gestaltet sich ein Überblick über die Zusammenhänge zwischen den einzelnen Projekten oft schwierig. Das Layer Diagram, das zu den Neuerungen im *Visual Studio 2010* gehört, ermöglicht eine Modellierung der Beziehungen zwischen Komponenten sowie eine konkrete Zuordnung von *Visual Studio*-Projekten zu diesen. Von der vorgegebenen Architektur abweichende Projektreferenzen („Abkürzungen“), d. h. Verstöße gegen die Schichtenarchitektur, werden (siehe Abbildung 2) sichtbar. Zusätzlich ist es möglich, den einzelnen Schichten Namensräume zuzuordnen. Auch hier werden fehlerhafte Zuordnungen schnell erkannt. Durch die Übersicht über das Gesamtprojekt bietet sich das Diagramm weiterhin als Einstieg in ein manuelles Review an.

Eine weitere Möglichkeit, insbesondere in komponentenorientierten Architekturen, für die Einhaltung von Komponenten- und Schichtgrenzen zu sorgen, ist die Einbindung von Metadaten auf Assemblyebene. Über diese kann einem Assembly dessen Rolle, z. B. Implementierung oder Schnittstellenbeschreibung innerhalb einer Komponente – im Beispiel „Versioning“ – sowie die Zugehörigkeit zu einem bestimmten Softwarelayer, z. B. Logic oder UserInterface, zugeordnet werden. Anhand dieser Informationen lassen sich mittels einfachen Regelwerks erlaubte und unerlaubte Abhängigkeitsbeziehungen definieren, die durch automatisierte Tests verifiziert werden können.

Darüber hinaus stellen Metriken ein weiteres Mittel der Architekturanalyse dar. Auf Basis von Kennzahlen wie z. B. der Vererbungstiefe, der Abhängigkeit von Klassen untereinander sowie der Komplexität von Funktionen wird, wie in Abbildung 4 dargestellt, ein Wartbarkeitsindex errechnet. Ausgehend von einer Kennzahl für einzelne Projekte kann in die Applikationsstruktur hinein navigiert und kritische Punkte somit identifiziert werden. Dies wird durch die farbliche Hervorhebung (Ampel) zusätzlich vereinfacht. Die Kennzahlen werden im Rahmen der Statischen Code-Analyse überwacht. Das Überschreiten von Grenzwerten wird durch die Ausgabe von Warnungen während des Kompilierens sichtbar. Die zu überwachenden Metriken lassen sich durch Verwendung von Tools wie NDepend oder RSM erweitern und in Form von Reports kontinuierlich überwachen.

Kontinuierliche Prüfung

Viele der vorgestellten Werkzeuge können in einen Buildprozess integriert werden. Ein automatisches Review kann somit zum Beispiel im Rahmen eines *Team Foundation Server* Daily Builds realisiert werden. Abbildung 5 zeigt die Ergebnisse der automatischen Analyse. Die ermittelten Kennzahlen werden in einem Data Warehouse hinterlegt. Reports (vgl. Abbildung 6) ermöglichen eine stetige Beurteilung der Quellcode-Qualität und erlauben eine sofortige Reaktion. Die Erfolge von Refactorings sowie Konsolidierungen im Rahmen von Stabilisierungsphasen werden ebenfalls sichtbar.

Sicherstellung der Einhaltung von Standards

Alle vorgestellten Werkzeuge hinterlegen ihre Konfiguration pro Solution oder pro Projekt. Eine Vereinheitlichung unternehmensweit, über mehrere Anwendungen hinweg, bedarf einer Synchronisierung von diesen Einstellungen. Eine Möglichkeit hierzu bietet der *Team Foundation Server* durch die

Code Metrics Results

Filter: None | Min: | Max:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Calculator (Debug)	100	6	1	1	0
Calculator.Model (Debug)	90	52	1	21	95
Calculator.Service (Debug)	85	127	4	44	246
{} AIT.Calculator.Scan	88	36	1	4	51
{} AIT.Calculator.Service	80	51	2	40	126
Calculator	70	7	1	7	18
CalculatorError	92	4	1	1	7
CalculatorException	92	6	2	2	8
CSharpParser	59	6	1	14	22
CSharpParser()	100	1		0	1
Parse(string) : IProcessorDa	51	5		13	21
CSharpProcessData	94	2	1	2	3

Abb. 4: Architektur- und Code-Metriken.

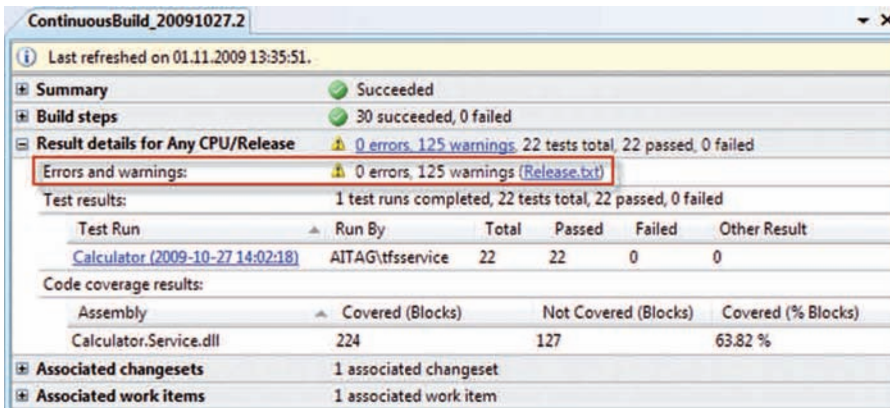

Abb. 5: Fehleranalyse im Rahmen des Buildprozesses.

Festlegung von Check-In Policies. Sind diese gesetzt, ist es nur noch möglich, Quellcode in das Repository einzuchecken, der den hinterlegten Richtlinien (Statische Code Analyse, *StyleCop*) genügt. Über diese Einstellungen hinaus geht das kostenlos zum Download bereitstehende *AIT Apply Company Policy Add-In* [AIT]. Dieses hilfreiche Add-In vereinheitlicht Einstellungen bezüglich der Dokumentation, den Pre- und Postbuild Events und dem Trace Level entsprechend der vom Unternehmen festgelegten Richtlinien.

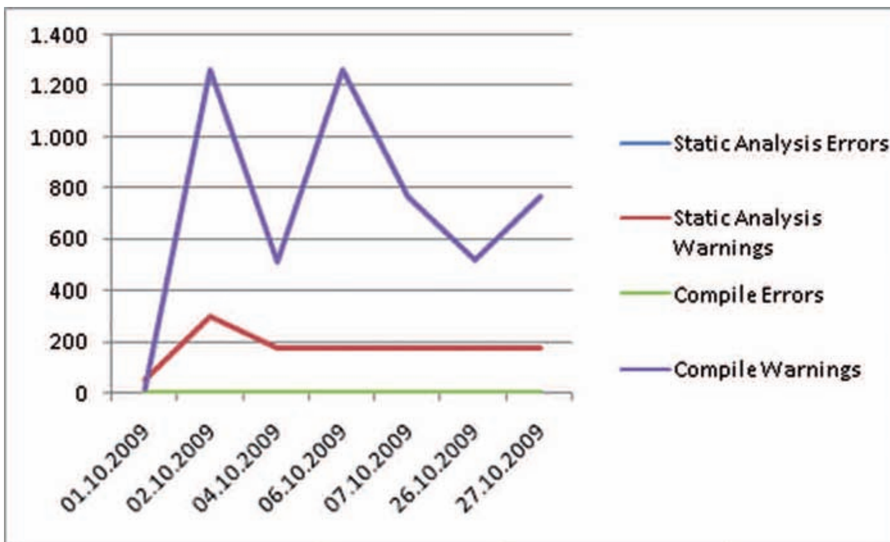

Abb. 6: Verlauf der Entwicklung von Warnungen und Fehlern.

Fazit

Review-Prozesse können zu einem großen Teil automatisiert werden. Viele Werkzeuge stehen kostenlos zur Verfügung und bedürfen nur einer minimalen Konfiguration. Somit kann bereits mit sehr geringem Aufwand die Qualität einer Applikation sowie die des zugrunde liegenden Quellcodes deutlich verbessert werden. Die Validierung der Programmierrichtlinien schafft dabei die Basis. Eine Überprüfung von Architekturrichtlinien baut auf dieser auf.

Die entstehenden Freiräume können für intensive Pair-Reviews genutzt werden. In diesen ist eine Konzentration auf die wesentlichen fachlichen und architekturbezogenen Aspekte möglich. Die Gesamtqualität der Anwendung kann damit zusätzlich entscheidend verbessert werden.

CODE-REVIEWS – ab wann profitieren auch Ihre Anwendungen davon? ■

Literaturverzeichnis

[StyMic] StyleCop. [Online] Microsoft. http://code.msdn.microsoft.com/sourceanalysis.

[AIT] AIT Apply Company Policy. [Online] AIT. http://www.aitag.com.